codingem.com

103 JavaScript Interview Questions & Answers (in 2023)

By Artturi Jalli

JavaScript is one of the most popular programming languages out there. It is used widely in the field of web development and it is an in-demand skill in the job market.

Here is a comprehensive list of 100 programming interview questions and answers. In addition to interviews, these can be handy if you're studying for a JavaScript examination.

Even if you are not participating in a programming interview or exam, this list is worthwhile—It goes through most of the important concepts of JavaScript.

1. What is JavaScript?

JavaScript is a client-side/server-side programming language. JavaScript can be inserted into HTML to make a web page interactive and enable user engagement.

2. What is hoisting in JavaScript?

Hoisting means all the declarations are moved to the top of the scope. This happens before the code is run.

For functions, this means you can call them from anywhere in the scope, even before they are defined.

```
hello();  // Prints "Hello world! " even though the function is called "befor

function hello(){
  console.log("Hello world! ");
}
```

For variables, hoisting is a bit different. It assigns `undefined` to them at the top of the scope.

For example, calling a variable before defining it:

```
console.log(dog);
var dog = "Spot";
```

Output:

```
undefined
```

This may be surprising because you probably expected it to cause errors.

If you declare a function or variable, it is always moved on the top of the scope no matter where you declare it.

3. What does the isNan() function do?

You can use `isNan()` function to check if the type of a value is a number and it's `NaN`.

(And yes, `NaN` is of type `number` even though it's a "Not-a-Number" by name. This is because it's a numeric type under the hood, even though it does not have a numeric value.)

For example:

```
function toPounds(kilos) {
  if (isNaN(kilos)) {
    return 'Not a Number! Cannot be a weight.';
  }
  return kilos * 2.2;
}

console.log(toPounds('this is a test'));
console.log(toPounds('100'));
```

Output:

```
Not a Number! Cannot be a weight.
220.00000000000003
```

4. What is negative infinity in JavaScript?

You get a negative infinity if you divide a negative number with zero in JavaScript.

For instance:

```
console.log(-10/0)
```

Output:

```
-Infinity
```

5. What is an undeclared variable? How about an undefined variable?

Undeclared variables do not exist in a program at all. If your program tries to read an undeclared variable, a runtime error is thrown.

An example of calling an undeclared variable obviously results in an error:

```
console.log(dog);
```

Output:

```
error: Uncaught ReferenceError: dog is not defined
```

Undefined variables are declared in the program but don't have a value. If the program tries to read an undefined variable, an undefined value is returned and the app does not crash.

An example of an undefined variable is:

```
let car;
```

```
console.log(car);
```

Output:

```
undefined
```

6. What types of popup boxes are there in JavaScript

The three types of popups are `alert`, `confirm`, and prompt. Let's see an example use of each:

Alert

For example:

```
window.alert("Hello, world!");
```

Confirm

For example:

```
if (window.confirm("Are you sure you want to go?")) {
  window.open("exit.html", "See you again!");
}
```

Prompt

For example:

```
let person = window.prompt("Enter your name");

if (person != null) {
  console.log('Hello', person);
}
```

7. What is the difference between == and ===?

- == compares values
- === compares both the value and the type

Example:

```
var x = 100;
var y = "100";

(x == y)  // --> true because the value of x and y are the same

(x === y) // --> false because the type of x is "number" and type of y is "st
```

8. What does implicit type coercion do? Give an example.

Implicit type coercion means a value is converted from one type to another under the hood. This takes place when the operands of expressions are of different types.

For example, string coercion means applying + operator on a number and a string converts the number into a string automatically.

For example:

```
var x = 1;
var y = "2";

x + y // Returns "12"
```

But when dealing with subtraction, the coercion works the other way. It converts a string into a number.

For instance:

```
var x = 10;
var y = "10";

x - y // Returns 0
```

9. Is JavaScript statically or dynamically typed language? What does this mean?

JavaScript is dynamically typed.

This means the type of objects is checked during run-time**.** (In a statically typed language, the type is checked during compile-time.)

In other words, JavaScript variables are not associated with a type. This means you can change the data type with no problem.

```
var num = 10;
num = "Test";
```

In a statically typed language, such as C++, changing an integer to a string this way is not possible.

10. What is NaN in JavaScript?

`NaN` means "Not-a-Number". This means a value that is not officially a number in JavaScript.

What may be confusing is that type-checking `NaN` with `typeof()` function results in `Number`.

```
console.log(typeof(NaN))
```

Output:

```
Number
```

To avoid confusion, use `isNaN()` to check if the type of a value is NaN or not number.

11. What is the spread operator in JavaScript?

Spread operator allows iterables (arrays / objects / strings) to be expanded into single arguments/elements.

Let's take an example to see this behavior in action:

```
function sum(a, b, c) {
  return a + b + c;
}
const nums = [15, 25, 35];
console.log(sum(...nums));
```

Output:

```
75
```

To learn more about the three dots operator (...) in JavaScript, feel free to read this article.

12. What is a closure in JavaScript?

A closure in JavaScript means an inner function has access to the variables of the outer function—even after the outer function has returned.

For example, to create a counter that increments itself by 1, you can use a closure:

```
function createCounter() {
   let counter = 0;
   function increment() {
     counter++;
     console.log(counter);
   }
   return increment;
}
```

Here `createCounter()` is the outer function and `increment()` is the inner one.

Now you can use it as follows:

```
const add = createCounter();

add(); // prints out 1
add(); // prints out 2
add(); // prints out 3
```

This works because `add` which stores the inner function `increment()` still has access to `counter` variable of the `createCounter()` function.

This is possible because of the closure nature of JavaScript: An inner function has access to the outer function's variables even after the outer function has returned.

13. How to handle exceptions in JavaScript?

If an expression **throws** errors, you can handle them with the **`try…catch`** statement.

The idea of using this structure is to try running an expression, such as a function with an input, and catch the possible errors.

For example:

```
function weekDay(dayNum) {
  if (dayNum < 1 || dayNum > 7) {
    throw 'InvalidDayNumber'
  } else {
    return ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun'][dayNum - 1];
  }
}

try { // Try to run the following
  let day = weekDay(8);
  console.log(day);
}
catch (e) { // catch an error if the above try failed
  let day = 'unknown';
  console.log(e);
}
```

14. What Is web storage?

Web storage is an API that provides a way for browsers to store key-value pairs to the user's browser locally. Using web storage makes this process way more intuitive than using cookies.

Web storage provides two ways for storing data:

- **Local storage** — stores data for the client without an expiration date.
- **Session storage** — stores data for only one session. The data is gone when the browser is closed.

Here's an example of how you can save, access, and delete an item from `sessionStorage`:

```
// Save data to sessionStorage
sessionStorage.setItem('favoriteColor', 'gray');

// Get the color from the sessionStorage
let data = sessionStorage.getItem('favoriteColor');
console.log(data);

// Remove saved color preset from sessionStorage
sessionStorage.removeItem('favoriteColor');

// Remove ALL the saved data from sessionStorage
sessionStorage.clear();
```

And here's how you can do the same using the `localStorage`:

```
// Save data to localStorage
localStorage.setItem('favoriteColor', 'gray');

// Get the color from the localStorage
let data = localStorage.getItem('favoriteColor');
console.log(data);

// Remove saved color preset from localStorage
localStorage.removeItem('favoriteColor');

// Remove ALL the saved data from localStorage
localStorage.clear();
```

15. Why do you need web storage?

Web storage (question 14) makes it possible to store big amounts of data locally. The key is it does not affect the performance of a website.

Using web storage, the information is not stored in a server. This makes it a more preferable approach compared to cookies.

16. What are modules?

Modules are units of reusable code. Usually, you can import a useful function or constructor to your project from the module.

Importing features from modules could look like this:

```
import { hello } from './modules/helloWorld.js';
```

17. What is meant by “scope” in JavaScript?

The scope defines the “visibility of your code”.

More formally, the scope describes where variables, functions, and other objects are accessible in the code. Scopes are created in your code during the runtime.

For example, a block scope means the “area” between curly braces:

```
if(true) {
   let word = "Hello";
}

console.log(word); // ERROR OCCURS
```

Here, the variable `word` is not accessible from anywhere else but in the if-statement.

18. What are higher-order functions in JavaScript?

A higher-order function operates on another function(s). It either takes a function as an argument or returns another function.

For instance:

```
function runThis(inputFunction) {
  inputFunction();
}

runThis(function() { console.log("Hello world") });
```

Output:

```
Hello world
```

Another example:

```
function giveFunction() {
  return function() {
    console.log("Hello world")
  }
}

var action = giveFunction();
action()
```

Output:

```
Hello world
```

19. What is the “this” keyword in JavaScript?

This refers to the object itself.

For example:

```
var student = {
    name:  "Matt",
    getName: function(){
      console.log(this.name);
    }
}

student.getName();
```

Output:

```
Matt
```

To make the `getName()` method work in the `student` object, the object has to access its own properties. This is possible via `this` keyword inside the object.

20. What does the .call() method do?

The `call()` method can be used to call a method of an object on another object.

```
obj1.func.call(obj2)
```

For example:

```
var student = {
  name: "Matt",
  getName: function(){
    console.log(this.name);
  }
}

var anotherStudent = {
  name: "Sophie"
};

student.getName.call(anotherStudent);
```

Output:

```
Sofie
```

`Call()` method can also be used to invoke a function by specifying the owner object.

For example:

```
function sayHi(){
  console.log("Hello " + this.name);
}

var person = {name: "Matt"};

sayHi.call(person);
```

Output:

```
Hello Matt
```

The `call()` can also accept arguments.

For example:

```
function sayHi(adjective){
  console.log("Hello " + this.name + ", You are " + adjective);
}

var obj = {name: "Matt"};

sayHi.call(obj, "awesome");
```

Output:

```
Hello Matt, you are awesome
```

21. What is the apply() method?

The `apply()` method does the same as the `call()` method. The difference is that `apply()` method accepts the arguments as an array.

For example:

```
const person = {
    name: 'John'
}

function greet(greeting, message) {
    return `${greeting} ${this.name}. ${message}`;
}

let result = greet.apply(person, ['Hello', 'How are you?']);

console.log(result);
```

Output:

```
Hello John. How are you?
```

In the line:

```
let result = greet.apply(person, ['Hello', 'How are you?']);
```

'Hello' is assigned to `greeting` and 'How are you?' is assigned to `message` in the `greet()` function.

22. What is the bind() method?

The `bind()` method returns a new function whose `this` has been set to another object.

Unlike `apply()` and `call()`, the `bind()` does not execute the function immediately. Instead, it returns a new version of the function whose `this` is set to another value.

Let's see an example:

```
let person = {
    name: 'John',
    getName: function() {
        console.log(this.name);
    }
};

window.setTimeout(person.getName, 1000);
```

This does **not** print the name "John", instead, it prints `undefined`.

To understand why this happens, re-write the last line in an equivalent way:

```
let func = person.getName;
setTimeout(func, 1000);
```

The `setTimeout()` receives the function **separately** from the `person` object but does not have the `person`'s name. Thus when `setTimeout()` invokes `person.getName`, the `name` is `undefined`.

To fix this, you need to bind the `getName()` method to the `person` object:

```
let func = person.getName.bind(person);
setTimeout(func, 1000);
```

Output:

```
John
```

Let's inspect how this approach works:

- The `person.getName` method is bound to the `person` object.
- The bound function `func` has now it's `this` value set to the `person` object. When you pass this new bound function to the `setTimeout()` function, it knows how to get the `name` of the `person`.

23. What is currying?

Currying means transforming a function with n arguments, to n functions of one or fewer arguments.

For example, say you have a function `add()` that sums up two numbers:

```
function add(a, b) {
    return a + b;
}
```

You can call this function by:

```
add(2,3)
```

Let's then curry the function:

```
function add(a) {
  return function(b) {
    return a + b;
  }
}
```

Now you can call this curried function by:

```
add(2)(3)
```

Currying does not change the behavior of a function. It changes the way it's invoked.

24. What is a promise in JavaScript?

A promise is an object that may produce value in the future.

A promise is always in one of the possible states: fulfilled, rejected, or pending.

Creating a promise looks like this:

```
const promise = new Promise(function(resolve, reject) {
    // implement the promise here
})
```

For example, let's create a promise that's resolved two seconds after being called.

```
const promise = new Promise(resolve => {
  setTimeout(() => {
    resolve("Hello, world!");
  }, 2000);
}, reject => {});
```

Now the key of promises is you can execute code right after the promise is resolved by using the `.then()` method:

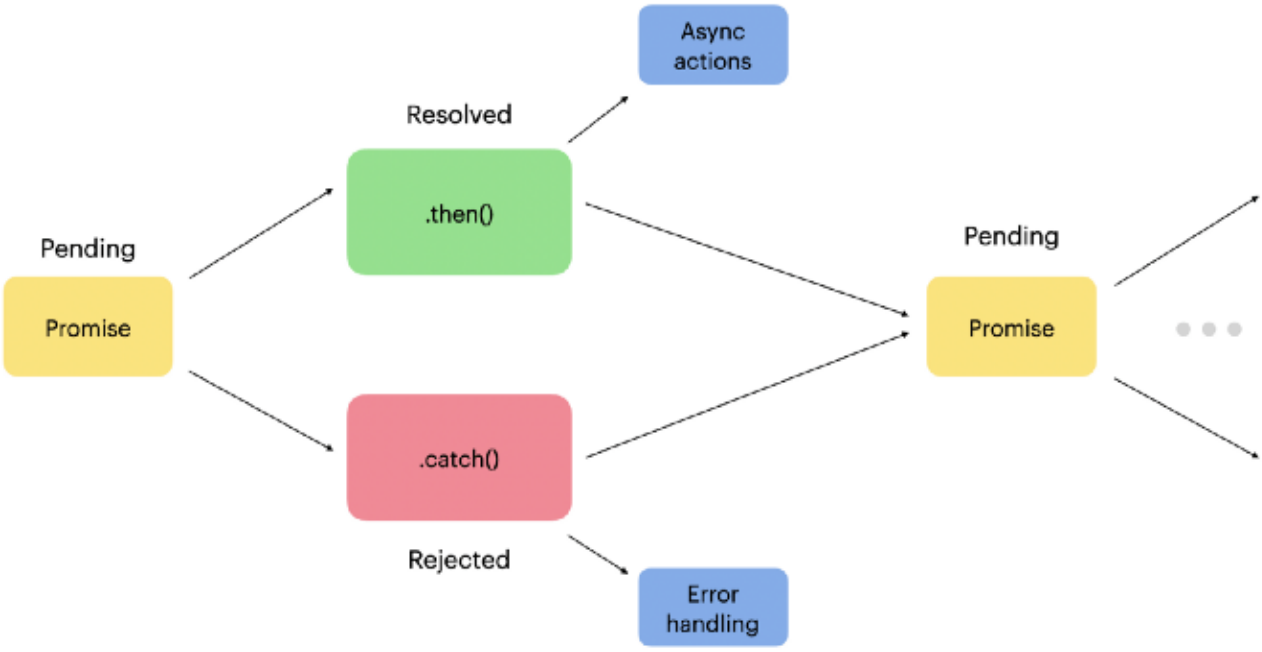
```
promise.then(result => console.log(result));
```

Output:

```
Hello, world!
```

Promises can be chained together such that a resolved promise returns a new promise.

Here is a flow chart of promises that also illustrates how they can be chained:



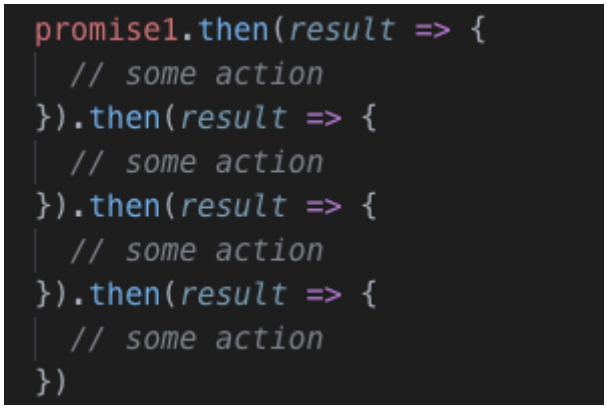
25. Why use promises?

In JavaScript, promises are useful with async operations.

In the past, one had to handle asynchronous operations with callbacks, that is, with functions that execute right after the operation has been completed. This caused a “callback hell”, a pyramid-shaped code with nested callbacks.

```
asyncOperation('something', function(error, data) {
  asyncOperation('something', function(error, data) {
    asyncOperation('something', function(error, data) {
      asyncOperation('something', function(error, data) {
        asyncOperation('something', function(error, data) {
          // Final action here...
        });
      });
    });
  });
});
```

Promises provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code. This is possible because promises can be chained in the following fashion:



```
promise1.then(result => {
  // some action
}).then(result => {
  // some action
}).then(result => {
  // some action
}).then(result => {
  // some action
})
```

26. What is a callback function in JavaScript?

A callback function is a function that is passed as an argument to another function. This function is executed inside the function to which it's passed as to "call back" when some action has been completed.

Let's see an example of this:

```
function greetName(name) {
  console.log('Hello ' + name);
}

function askName(callback) {
  let name = prompt('Enter your name.');
  callback(name);
}

askName(greetName);
```

This code prompts you a name, and when you've entered the name, it says "Hello" to that name. So the callback function, which in this case, is the `greetName`, executes only after you've entered a name.

27. Why use callbacks in JavaScript?

The callbacks are useful because JavaScript is an event-driven language. In other words, instead of waiting for a response, it keeps executing while listening for other events.

The above example demonstrates the usefulness of callbacks in JavaScript:

```
function greetName(name) {
  console.log('Hello ' + name);
}

function askName(callback) {
  let name = prompt('Enter your name.');
  callback(name);
}

askName(greetName);
```

28. What is a strict mode in JavaScript?

Strict mode allows you to set your program to operate in a strict context.

The strict mode prevents certain actions from being taken. Also, more exceptions are thrown.

The expression `"use strict";` tells the browser to enable the strict mode.

For example:

```
"use strict";
number = 1000;
```

This causes an error because strict mode prevents you from assigning a value to an undeclared variable.

29. What is an Immediately Invoked Function?

An Immediately Invoked Function (IIFE) runs right after being defined.

For example:

```
(function(){
  // action here
})();
```

To understand how an IIFE works, look at the parenthesis around it:

- When JavaScript sees the keyword `function`, it assumes there's a function declaration coming.
- But the declaration above is invalid because the function does not have a name.
- To fix this, the first set of parenthesis around the declaration is used. This tells the interpreter that it's a function expression, not a declaration.

```
(function (){
   // action here;
})
```

- Then, to call the function, another set of parenthesis needs to be added at the end of the function declaration. This is similar to calling any other function:

```
(function (){
  // action here
})();
```

30. What is a cookie?

A cookie is a small data packet stored on your computer.

For instance, a website can place cookies on visitors' browsers to remember login credentials the next time the user visits the page.

Under the hood, cookies are text files with key-value pairs. To create, read, or delete cookies, use the `document.cookie` property.

For example, let's create a cookie that saves the username:

```
document.cookie = "username=foobar123";
```

31. Why use strict mode in JavaScript?

Strict mode helps writing “secure” JavaScript code. This means bad syntax practices are converted into real errors.

For example, strict mode prevents creating global variables.

To declare strict mode, add `‘use strict’;` statement before the statements which you want to be under strict mode:

```
'use strict';
const sentence = "Hello, this is very strict";
```

32. What does double exclamation do?

The double exclamation converts anything to boolean in JavaScript.

```
!!true    // true
!!2       // true
!![]      // true
!!"Test"  // true

!!false   // false
!!0       // false
!!""      // false
```

This works because anything in JavaScript is “Truthy” or “Falsy” by nature.

33. How can you delete a property and its value?

You can use the `delete` keyword to delete the property and its value from an object.

Let’s see an example of this:

```
var student = {name: "John", age:20};
delete student.age;

console.log(student);
```

Output:

```
{name: "John"}
```

34. How can you check the type of a variable in JavaScript?

Use the `typeof` operator.

```
typeof "John Abraham"  // Returns "string"
typeof 100             // Returns "number"
```

35. What is null in JavaScript?

`null` represents the absence of value. It highlights that a variable points to no object.

The type of `null` is an `object`:

```
var name = null;
console.log(typeof(name))
```

36. Null vs undefined?

Null and undefined are easy to get confused with.

However, it is important to understand what is the difference between the two.

Null

In JavaScript, the `null`:

- Is a value that indicates the variable points to no object.
- Is of type `object`.
- Represents null, empty, or non-existent reference.
- Represents the absence of a value of a variable.
- Converts to `0` with primitive operations.

Undefined

The `undefined`:

- Is a value that represents a variable that has been declared but doesn't have a value
- Is of type `undefined`.
- Represents the absence of the variable.
- Converts to `NaN` with primitive operations.

37. Can you access history in JavaScript?

Yes, it is possible.

You can access history via `window.history` that contains the browser's history.

To retrieve the previous and next URLs, use these methods:

```
window.history.back();
window.history.forward();
```

38. What is a global variable?

A global variable is available everywhere in the code.

To create a global variable, omit the var keyword:

```
x = 100; // Creates a global variable.
```

Also, if you create a variable using `var` outside any functions, you create a global variable.

39. Does JavaScript relate to Java?

Nope.

They are two different programming languages that have nothing to do with one another.

40. What are JavaScript events?

Events are what happens to HTML elements. When JavaScript is used in an HTML page, it can react to the events, such as a button click.

Let's create an HTML page where is a button, and when that button is clicked, an alert shows up:

```
<!doctype html>

<html>
 <head>
   <script>
     function sayHi() {
       alert('Hi, how are you?');
     }
   </script>
 </head>

 <body>
   <button type="button" onclick="sayHi()">Click here</button>
 </body>

</html>
```

41. What does the preventDefault() method do?

The `preventDefault()` cancels a method. The name `preventDefault` describes the behavior well. It prevents the event from taking the default behavior.

For example, you can prevent a form submission when submit button is clicked:

```
document.getElementById("link").addEventListener("click", function(event){
  event.preventDefault()
});
```

42. What is the setTimeout() method?

The `setTimeout()` method calls a function (once) after a specified number of milliseconds. For example, let's log a message after one second (`1000ms`):

```
setTimeout(function() {
    console.log("Good day");
}, 1000);
```

43. What is the setInterval() method?

The `setInterval()` method calls a function periodically with a custom interval.

For example, let's periodically log a message every second:

```
setInterval(function() {
    console.log("Good day");
}, 1000);
```

44. What is ECMAScript?

ECMAScript is the scripting language that forms the basis of JavaScript.

ECMAScript is standardized by the ECMA International standards organization (check out ECMA-262 and ECMA-402 specifications).

45. What is JSON?

JSON (JavaScript Object Notation) is a lightweight data format used to exchange data.

For example, here's a JSON object:

```
{
    'name': 'Matt',
    'address': 'Imaginary Road 22',
    'age': 32,
    'married': false,
    'hobbies': ['Jogging', 'Tennis', 'Padel']
}
```

The syntax rules for JSON are:

1. The data is in key-value pairs.
2. The data is separated by commas.
3. Curly braces define an object.
4. Square brackets define an array.

In case you're interested, here is a complete guide to what is JSON.

46. Where is JSON used?

When sending data to a server and vice versa, the data must be in a text format.

JSON is a text-only format that allows sending data to the server and data from the server to a browser. JSON is supported by nearly all programming languages so it can be used with other languages too.

47. Why use JSON stringify?

When you send data to a server, it has to be a string.

To convert a JavaScript object into a string, you can use the `JSON.stringify()` method.

```
var dataJSON = {name: "Matt", age: 51};
var dataString = JSON.stringify(dataJSON);

console.log(dataString);
```

Output:

```
'{"name":"Matt","age":51}'
```

48. How can you convert JSON string to a JSON object?

When you receive data from a server, it's always in a string format. To convert a JSON string to a JavaScript object, use the `JSON.parse()` method.

```
var data = '{"name":"Matt", "age":51}';
var dataJSON = JSON.parse(data);console.log(dataJSON);
```

Output:

```
{
    name:"Matt",
    age:51
}
```

49. How can you assign a default value to a variable?

Use a logical operator || in the assignment to provide a default value.

```
const a = b || c;
```

This works such that if b is falsy then c is going to be assigned to a. (Falsy means `null`, `false`, `undefined`, `0`, empty string, or `NaN`.)

50. Can you define properties for a function?

Yes, because functions are also objects.

Let's see an example of this:

```
let func = function(x) {

};

func.property1 = "Hello there";

console.log(func.property1);
```

Output:

```
Hello there
```

51. What is the meaning of a race method in promises?

`Promise.race()` method returns the promise which is resolved or rejected first.

Let's demonstrate this with an example of where the promise number two resolves faster than the first one:

```
let p1 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 500, 'the first promise');
});

let p2 = new Promise(function(resolve, reject) {
    setTimeout(resolve, 100, 'the second promise');
});

Promise.race([p1, p2]).then(function(value) {
  console.log(value, 'was faster');
});
```

Output:

```
the second promise was faster
```

52. What does promise.all() method do?

`Promise.all` is a promise that takes an array of promises as input. It gets resolved when:

- Either all the input promises get resolved.
- Or any one of them gets rejected.

For instance, `promise.all` waits for all these three promises to complete:

```
var prom1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Yay!");
  }, 1000);
});
```

```
var prom2 = Promise.resolve(10);
var prom3 = 100;

Promise.all([prom1, prom2, prom3]).then(values => {
  console.log(values);
});
```

Output after one second has passed:

```
["Yay", 10, 100]
```

53. What is the eval() function?

The `eval()` function evaluates code inside a string. The string to be evaluated can be an expression, variable, statement, or sequence of statements.

For example:

```
console.log(eval("5+10"));
```

Output:

```
15
```

54. What is event bubbling?

In event bubbling an event starts by running the event handlers on the innermost element. Then it triggers the parents' event handlers until it reaches the outermost element.

The best way to see this in action is by creating an HTML document with divs inside of divs:

```
<style>
  body * {
    margin: 20px;
    border: 1px solid blue;
  }
```

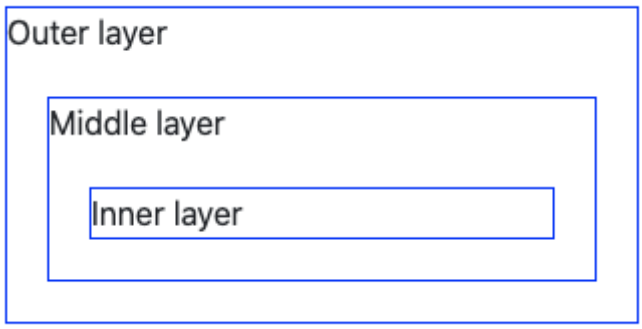
```
</style>

<div onclick="alert('Outer layer')">Outer layer
  <div onclick="alert('Middle layer')">Middle layer
    <div onclick="alert('Inner layer')">Inner layer

    </div>
  </div>
</div>
```

In each `div`, there's a JavaScript `alert` that's triggered when one clicks the div.

The result page looks like this:



If you now click the `Inner layer`, it triggers the `alert` assigned to that div, and it triggers the alerts of the parent divs.

55. What is the Temporal Dead Zone?

Temporal Dead Zone means unreachability of a variable, even though it's already in the scope.

Let's first take a look at what happens when you try to log a variable to the console when it's not initialized:

```
console.log(x);
var x = "Yay";
```

Output:

```
undefined
```

You probably expected this to result in an error, but it prints `undefined`.

This happens because of hoisting, which means all the declarations are moved to the top of the scope. Due to hoisting, the above code behaves like this under the hood:

```
var x;

console.log(x);

x = "Yay";
```

Here `undefined` is assigned to the variable at the top automatically. This makes it possible to use it before defining it.

But let's then take a look at what happens when we do the same using `let` instead of `var`:

```
console.log(x);
let x = 10;
```

Output:

```
error: Uncaught ReferenceError: Cannot access 'x' before initialization
```

This happens because `let` is hoisted differently than `var`.

When a `let` variable is hoisted it doesn't become `undefined`. Instead, it's unreachable, or in the Temporal Dead Zone until it's assigned a value.

56. What is URI?

A URI or Uniform Resource Identifier is a set of characters that distinguish resources from one another. URIs allow internet protocols to perform the actions between resources.

A URI can look something like this:

```
hello://example.com:8042/there?name=jack#sumthing
```

57. What is the DOM?

When a web page has loaded, the browser creates a DOM for the page. This gives JavaScript the power to create dynamic HTML.

DOM or Document Object Model acts as an API for HTML documents. It defines the structure of the documents. It also specifies how the document is accessed and modified.

58. Document load vs DOMContentLoaded?

- The `DOMContentLoaded` event is triggered when the HTML document has been loaded and parsed. It does not wait for the assets (such as stylesheets and images).
- The document load event triggers only after the full page has loaded, including all the assets.

For example, here's how you can use DOMContentLoaded to notify when the DOM has fully loaded:

```
window.addEventListener('DOMContentLoaded', (event) => {
    console.log('DOM is now loaded!');
});
```

And here's an example of how you can add a listener for when a specific page has loaded:

```
window.addEventListener('load', (event) => {
  console.log('The page is now loaded!');
});
```

59. HTML Attribute vs DOM Property?

When you write HTML you can define attributes on the HTML elements. Then, when you open the page with a browser, your HTML code is parsed. At this point, a DOM node is

created. This DOM node corresponds to the HTML document you've just written. This DOM node is an object that has properties.

For instance, this HTML element:

```
<input id="my-input" type="text" value="Name:">
```

has three attributes, `id`, `type`, and `value`.

When the browser parses this HTML element

- It takes this `input` field and bakes an `HTMLInputElement` object from it.
- This object has dozens of properties such as `accept`, `accesKey`, `align`.
- It also has some of the original HTML attributes turned into properties, such as the `id` and the `type`. But for example, the `value` property does not refer to the `value` attribute.

60. What is the same-origin policy?

The same-origin policy is a valuable security mechanism. It prevents JavaScript from making requests over the domain boundaries.

An origin refers to the URI scheme, hostname, and port number. The same-origin policy makes it impossible for a script on one page from obtaining access to sensitive data on another.

61. Is JavaScript a compiled or interpreted language?

JavaScript is an interpreted language.

An interpreter in the browser reads the JavaScript code, interprets each line, and runs it.

62. Is JavaScript a case-sensitive language?

JavaScript is a case-sensitive language.

The keywords, variables, function names, and so on need to be capitalized consistently.

To demonstrate, this piece of code works

```
let i = 1;

while(i < 2) {
  console.log(i);
  i++;
}
```

But this does not, because the `while` is capitalized even though it shouldn't.

```
let i = 1;

WHILE(i < 2) {
  console.log(i);
  i++;
}
```

63. How many threads are there in JavaScript?

JavaScript works with a single thread. It does not allow to write code that the interpreter could run in parallel in multiple threads or processes.

This means it executes code in order and must finish executing a piece of code before moving to the next one.

A good example of seeing this in action is when you show an alert on a web page. Once the alert pops up, you cannot interact with the page until the alert is closed.

```
alert("Hello there!");
```

64. What does the "break" statement do?

The `break` statement jumps out of a loop and continues executing the code outside of the loop.

For example, this loop terminates after it encounters a number 5:

```
for (var i = 0; i < 100; i++) {
  if (i === 5) {
    break;
  }
  console.log('Number is ', i);
}

console.log('Yay');
```

Output:

```
Number is 0
Number is 1
Number is 2
Number is 3
Number is 4
Yay
```

65. What does the “continue” statement do?

The `continue` statement jumps over one round of a loop.

For example, this loop skips numbers 2 and 3:

```
for (var i = 0; i < 5; i++) {
  if (i === 2 || i === 3) {
    continue;
  }
  console.log('Number is ', i);
}
```

Output:

```
0
1
4
```

66. What is a regular expression?

A regular expression, also known as regex or regexp, is a group of characters that form a search pattern. It's a pattern-matching tool that is commonly used in JavaScript and other programming languages.

For example, let's find any numbers from a string using a regular expression:

```
var regex = /\d+/g;
var string = "You have 100 seconds time to run";

var matches = string.match(regex);

console.log(matches);
```

Output is an array of all the matches:

```
[100]
```

For example, regex can be used to search for emails or phone numbers in a large text file.

67. What is the purpose of a breakpoint when debugging code?

Breakpoints allow you to find bugs in your JavaScript code.

You can set breakpoints in your code when a debugger statement is executed and the debugger window appears.

At a breakpoint, JavaScript stops executing and lets you inspect the values and the scope to solve possible issues.

68. What is a conditional operator?

A **conditional operator** is a shorthand for writing if-else statements. A conditional operator is sometimes known as a ternary operator.

For example:

```
// Regular if-else expression:
const age = 10;
if(age < 18){
  console.log("Minor");
} else {
  console.log("Adult");
}

// Conditional operator shorthand for the above if-else
age < 18 ? console.log("Minor") : console.log("Adult")
```

69. Can you chain conditional operators?

Yes, it is possible. Sometimes it can be useful as it can make the code more understandable.

Let's see an example of this:

```
function example() {
    if (condition1) { return value1; }
    else if (condition2) { return value2; }
    else if (condition3) { return value3; }
    else { return value4; }
}

// Shorthand for the above function
function example() {
    return condition1 ? value1
         : condition2 ? value2
         : condition3 ? value3
         : value4;
}
```

70. What does the freeze() method do?

The `freeze()` method freezes an object. It makes an object immutable.

After freezing an object, it's not possible to add new properties to it.

For example:

```
const item = { name: "test" };

Object.freeze(item);
item.name = "Something else"; // Error
```

71. How can you get the list of keys of an object?

Use the `Object.keys()` method.

For example:

```
const student = {
  name: 'Mike',
  gender: 'male',
  age: 23
};

console.log(Object.keys(student));
```

Output:

```
["name", "gender", "age"]
```

72. What are JavaScript's primitive data types?

A primitive data type has a primitive value. There are seven different primitive data types in JavaScript:

1. `string`— words. For example "`John`".
2. `number` — numeric values. For example 12.
3. `boolean` — true or false. For example `true`.
4. `null` — absence of a value. For example `let x = null;`
5. `undefined` —type where a variable is declared but does not have a value. For example, when creating variable x this way `let x;` , x becomes `undefined`.
6. `bigint` — An object meant to represent whole numbers greater than 2^53–1. For example `BigInt(1210313934547202 92)`

7. `symbol` — A built-in object for creating unique symbols. For example `let sym1 = Symbol('test')`

73. What ways are there to access properties of an object?

There are three ways to access properties.

- The dot notation.

For example:

```
obj.property
```

- The square brackets notation.

For example:

```
obj["property"]
```

- The expression notation.

For example:

```
obj[expression]
```

74. How to execute JavaScript code after the page has loaded?

There are three ways you can do this:

- Set the property `window.onload` as a function that executes after the page loads:

```
window.onload = function ...
```

- Set the property `document.onload` as a function that executes after the page loads:

```
document.onload = function ...
```

- set the `onload` property of an HTML attribute as a JS function:

```
<body onload="script();">
```

75. What is an error object?

An **error object** is a built-in object that gives you detailed error info if an error happens to occur.

The error object has two properties:

- `name`
- `message`

For example, let's assume that the `sayHi()` function throws an error. When this happens, the `catch` block gives you an `error` object you can print to the console for example.

```
  sayHi("Welcome");
}
catch(error) {
  console.log(error.name + "\n" + error.message);
}
```

76. What does the NoScript tag do?

Noscript tag is for detecting and reacting to browsers that have JavaScript disabled.

You can use the NoScript tag to execute a piece of code that notifies the user.

For example, your HTML page can have a `noscript` tag like this:

```
<script>
  document.write("Hello World!");
</script>

<noscript>
  Your browser does not support JavaScript!
</noscript>
```

77. What is an entry controlled loop?

In an entry-controlled loop, the condition is tested before entering the body of the loop.

For example, for loops and while loops fall into this category:

```
let nums = [1,2,3];

for (let num of nums) {
    console.log(num);
}
```

Output:

```
1
2
3
```

78. What is an exit controlled loop?

In an exit controlled loop, a condition is evaluated at the end of the loop. This means the loop body executes at least once regardless of whether the condition was `true` or `false`.

For example, the do-while loop falls into this category:

```
const i = 0;
```

```
do {
  console.log('The number is', i);
} while (i !== 0);
```

Output:

```
The number is 0
```

79. What is an anonymous function?

An anonymous function is a function that does not have a name.

Anonymous functions are commonly assigned to a variable name or used as a callback function.

Here is a function with a name for a reference:

```
function example(params) {
    // do something
}
```

Here is an anonymous function that's assigned to a variable:

```
const myFunction = function() {
    // do something
};
```

And here's an anonymous function used as a callback:

```
[1, 2, 3].map(function(element) {
     // do something
});
```

Learn more about callbacks and the map() function here.

80. What is an iterator?

The **iterator** protocol makes it possible for an object to generate a sequence of values.

The iterator must implement the `next()` method for getting the next value in the sequence. This method returns an object with

- `value`— The next value in the iteration sequence
- `done`—If this value is the last in the sequence, this is `true`. If not, then it's `false`.

Here is an example of creating and using an iterator. This function implements a range iterator that can be called by `rangeIter(1,5)`, and prints values `1 2 3 4`.

```
// define a function that returns an iterator
function rangeIter(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let count = 0;
    // create the actual iterator object
    const iterator = {
       // create the next() method that knows how to get the next value in th
       next: function() {
           let result;
           if (nextIndex < end) {
               // Return the value and set done 'false' because the iteratior
               result = { value: nextIndex, done: false }
               nextIndex += step;
               count++;
               return result;
           }
           // set done 'true' when the end has been reached
           return { value: count, done: true }
       }
    };
    // return an iterator object
    return iterator;
}

// Using the iterator
const it = rangeIter(1, 5);

let result = it.next();
while (!result.done) { // prints 1 2 3 4
     console.log(result.value);
     result = it.next();
}
```

81. What is an iterable?

The iterable protocol means that an object can be iterated over, and thus implements the iterator protocol (question 80.)

In other words, you can use the `for...of` loop on any iterable to loop through the sequence of values it generates.

For example, an `Array` or `Map` are iterables in JavaScript, but an `Object` is not.

Here is an example of applying a `for…of` loop on an array, which is iterable by nature:

```
const nums = [1,2,3];

for (let num of nums) {
    console.log(num);
}
```

Output:

```
1
2
3
```

82. What is a generator?

The generator is an alternative for iterators. You can write iterative code with non-continuous execution. In other words, the execution of a generator function can be paused.

Generators are defined using `function*` syntax. Instead of returning values, they `yield` values.

When created, generators do not execute their code. Instead, they return a Generator object, which is essentially an iterator. When you call `next()` on the generator object, it runs the code until it encounters a `yield` statement, and stops.

For example, here's a generator that does the exact same as the iterator in the iterator part above:

```
// Create a generator function that returns an iterator
function* rangeIter(start = 0, end = Infinity, step = 1) {
    let count = 0;
    for (let i = start; i < end; i += step) {
        count++;
        yield i;
    }
    return count;
}


// Create a generator object
const it = rangeIter(1, 5);


// Use the generator exactly how you'd use an iterator
let result = it.next();
while (!result.done) { // prints 1 2 3 4
     console.log(result.value);
     result = it.next();
}
```

The `rangeIter` function is much easier to read than the `rangeIter` in the iterator example. Yet both do the exact same thing.

83. What is a for of loop?

For…of loop can be used to iterate over iterables in JavaScript.

For example, you can print the contents of an array using a for…of loop:

```
const nums = [1,2,3];

for (const num of nums) {
    console.log(num);
}
```

Output:

```
1
2
3
```

84. What is nodejs?

Node. js is a popular open-source development platform.

It is used for executing JavaScript code server-side.

Node.js is handy for developing apps that need a persistent connection from the browser to the server.

Node.js is commonly used to build real-time apps such as a chat, news feed, or similar.

85. What is an event loop?

An event loop is a queue of callback functions. It handles all the asynchronous callbacks.

When an asynchronous function executes, a callback function is pushed into the queue. The JavaScript engine doesn't trigger the event loop before the async task has finished.

For example, the structure of an event loop could look like this:

```
while (queue.waitForMessage()) {
    queue.processNextMessage();
}
```

86. What is a Unary Operator?

The unary + operator is used to convert a variable to a number.

If the variable cannot be converted, it is converted to `NaN` (which is a special case of being a `number`, so the type is still a `number`).

For example:

```
let str = "10";
let num = +str;

console.log(typeof str, typeof num);

let word = "Hello";
let n = +word;

console.log(typeof word, typeof n, n);
```

Output:

```
string, number
string, number, NaN
```

87. How to sort elements of an array?

Use the `sort()` to sort the items of an array. This does not create a new array but sorts the original array “in-place”, that is, modifies it directly.

```
let months = ["Adam", "Sam", "Jack", "Bill"];
months.sort();

console.log(months);
```

Output:

```
["Adam", "Bill", "Jack", "Sam"]
```

88. What is TypeScript?

TypeScript is JavaScript with types. It’s a superset of JavaScript created by Microsoft.

TypeScript adds types such as optional types, classes, async/await, and so on to plain JavaScript.

Here is a simple example of a TypeScript function:

```
function greet(name: string): string {
   return "Hello, " + name;
}
```

The type information is explicitly expressed in the function's parameter type and return type.

89. What is a constructor in JavaScript?

A constructor is a method for creating and initializing a class object. It's executed when you instantiate a new object from a class.

For example:

```
class Student {
  constructor() {
    this.name = "Mike";
  }
}

let student = new Student();
console.log(student.name);
```

Output:

```
Mike
```

90. What is ES6?

ES6 (ECMAScript 6) is the sixth version of the JavaScript programming language.

The ES6 was released back in June 2015.

91. What are template literals?

Template literals allow you to embed expressions directly inside a string.

- When using a template literal, do not declare the string with quotation marks, but use backtick instead (`).
- To embed a variable or expression into the string, you need to add it between `${}`

For example:

```
console.log(`This is the ${10 * 10}th time`)
```

Output:

```
This is the 100th time
```

92. How can you swap two variables without a third?

Use destructuring to pull apart values from an array. This can also be used to swap two variables without a third helper:

```
let a = 1;
let b = 2;

[a, b] = [b, a];

console.log(a, b)
```

Output:

```
2 1
```

93. What is an ArrayBuffer?

An `ArrayBuffer` is a generic and fixed-length binary data buffer.

For instance:

```
let buffer = new ArrayBuffer(16);
```

```
console.log(buffer.byteLength)
```

Output:

```
16
```

94. What is a prototype?

All JavaScript objects inherit properties from a prototype.

For example:

- Math objects inherit properties from the `Math` prototype
- Array objects inherit properties from the `Array` prototype.

A prototype is a characteristic of an object. It describes the attributes associated with it. It acts as a blueprint of an object.

You can for example access an object's `prototype` to add a new property to an object constructor for example:

```
function Fruit(name, weight) {
  this.name = name;
  this.weight = weight;
}

Fruit.prototype.description = "Yum!";
```

95. What are arrow functions?

Arrow functions provide a shorthand for creating functions in JavaScript.

You can use arrow functions in function expressions only.

Here's a comparison of a regular function and an arrow function:

```
// Traditional function
var sum = function(a,b){
```

```
  return a + b;
}

// Arrow Function
var sum = (a,b) => a + b;
```

Let's examine it a bit more:

- Arrow functions are declared without the `function` keyword.
- If there is only one (returning) expression, you don't need to use the `return` keyword.
- In the above, also the curly braces are missing. This is only possible if the arrow function consists of only one expression. If there's more, then you need to add curly braces after the arrow.

96. What is the use of dir() method?

The `console.dir()` displays an interactive list of the properties of a JavaScript object as JSON.

For example:

```
const student = { "name":"Mike", "id": 132123, "city": "New York"};
console.dir(student);
```

Results in a following interactive list in the console:

97. How do you disable right-click on the web page?

You can disable the right click on a webpage by returning `false` from the `oncontextmenu` attribute on the `body` element.

```
<body oncontextmenu="return false;">
```

98. What is a unary function?

A unary function is a function that accepts only one argument.

For example:

```
function greet(name){
    console.log('Hello', name);
}
```

99. What is a pure function?

A pure function is a function that returns the same result with the same arguments regardless of when and where it's called. A function is pure if it does not depend on the state, or data change during a program's execution.

For example, a function that calculates the area of a circle is pure:

```
function circleArea(radius) {
  return Math.PI * Math.pow(radius, 2);
}
```

100. What is object destructuring?

Object destructuring is a way to extract properties from an object (or an array).

Before ES6 you would have needed to do this to extract the properties of an object:

```
const PersonDetails = {
  name: "Matty",
  age: 42,
  married: false
}

const name = PersonDetails.name;
const age = PersonDetails.age;
const married = PersonDetails.married;
```

```
console.log(name);
console.log(age);
console.log(married);
```

But since ES6 you could do that with one line of code by utilizing object destructuring:

```
const PersonDetails = {
  name: "Matty",
  age: 42,
  married: false
}

const {name, age, married} = PersonDetails;

console.log(name);
console.log(age);
console.log(married);
```

101. What Is an API?

An API stands for Application Programming Interface.

APIs are used to allow communication between different applications and parts of the application.

API is not a server. It is an intermediary piece of code that is used to access a server. In other words, API is an access point to a database.

You will use APIs a lot in web development.

For instance, you can build a weather website that queries weather data from a weather API.

To learn more about APIs, check out this guide.

102. What Is a Lexical Environment?

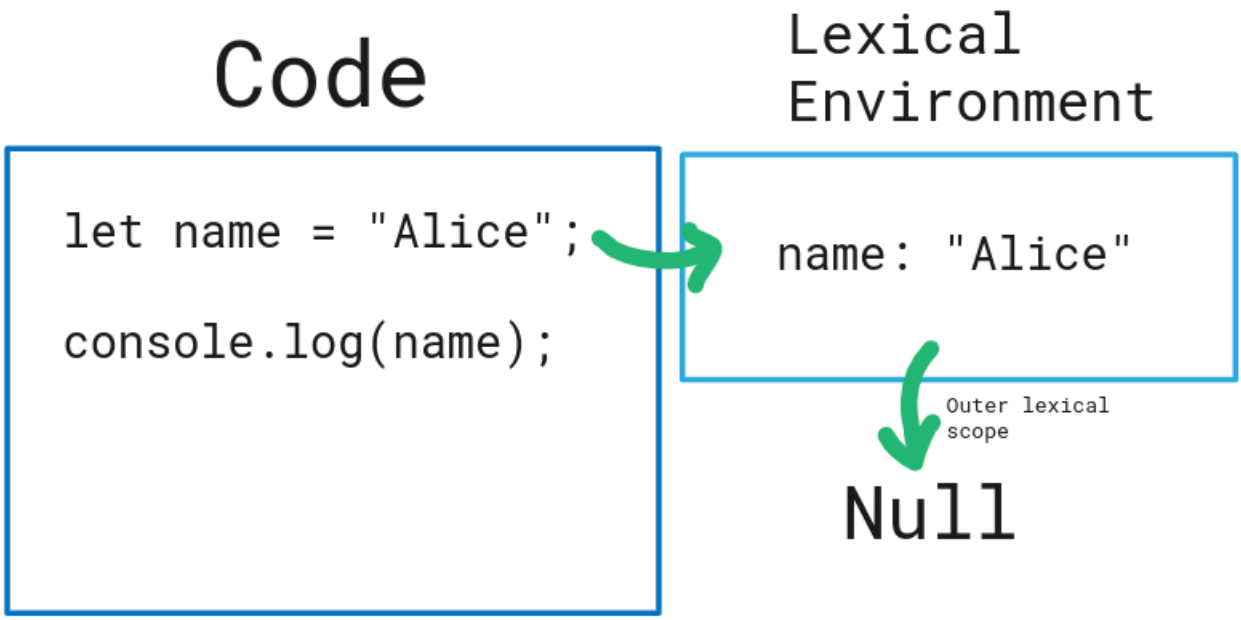
In JavaScript, each block of code is associated with a lexical environment behind the scenes.

It is an implementation detail of JavaScript.

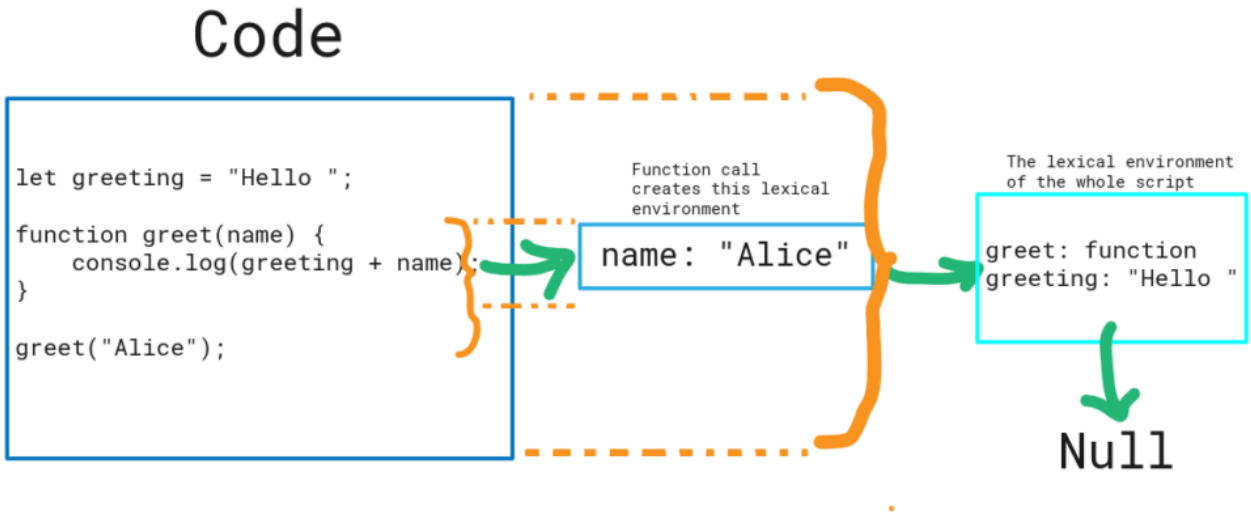
A lexical environment stores the variables of the block of code. The environment also stores a reference to a (possible) outer lexical environment. If there is no outer lexical environment, the environment reference is **null**.

When you modify a variable, you are essentially modifying a property of an environment object behind the scenes.

Here is an illustration of the global lexical environment of a script:



Each new block of code introduces a new inner lexical environment.



When a block of code is trying to access a variable, the lexical environments are searched recursively behind the scenes. If the variable is not found in the environment of the function, then the enclosing environment is searched next. This is continued until the global environment is searched.

This makes it possible to use variables declared outside of a function inside the function.

103. What Is the Epoch Time?

The Epoch time is the number of seconds since Jan 1, 1970.

This is also called the Unix timestamp.

An epoch time is necessary because computers need to have a starting point in time. Otherwise, they would not be able to tell the time.

Currently, the Epoch time is around 1.6 billion seconds.

Sometimes you see servers store timestamps as epoch times.

```
{
    "item": "T-shirt",
    "last_modified": 1647425869
}
```

Read more about epochs in computer science here.

Conclusion

Thanks for reading. I hope you enjoy it.

I wish you the best of luck with your interview or examination.

Happy coding!

← Previous Post Next Post →

About the Author

Hi, I'm Artturi Jalli! ✌️

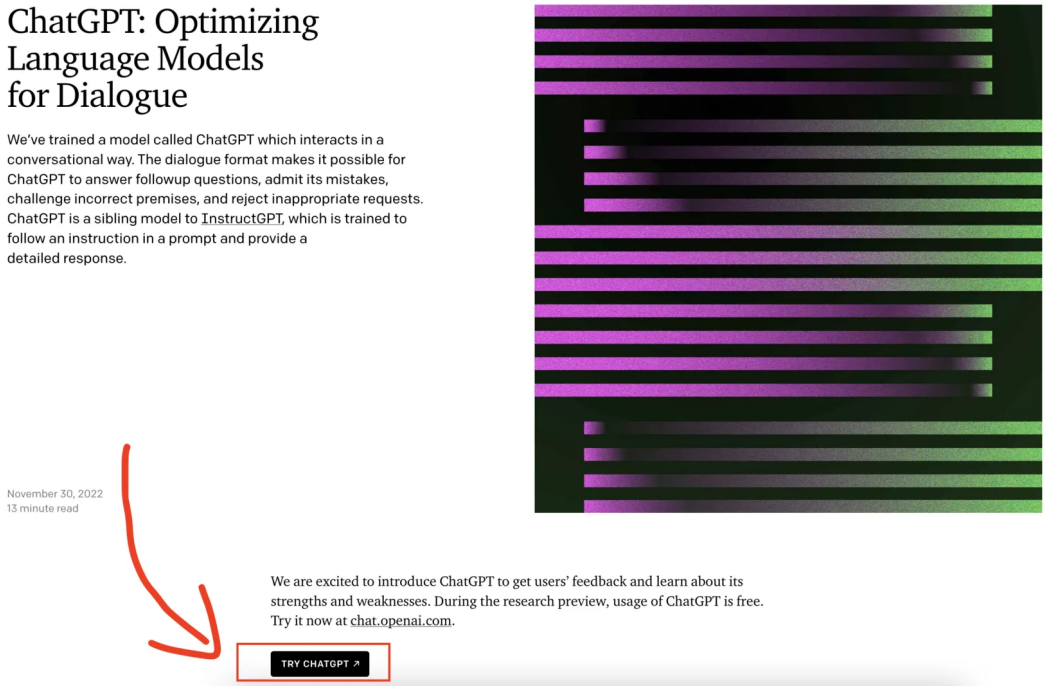
I'm a Tech enthusiast from Finland. 🇫🇮

👨‍🏫 I make Coding & Tech easy and fun with well-thought how-to guides and reviews.

🌍 I've already helped **3M+ visitors** reach their goals!

Contact

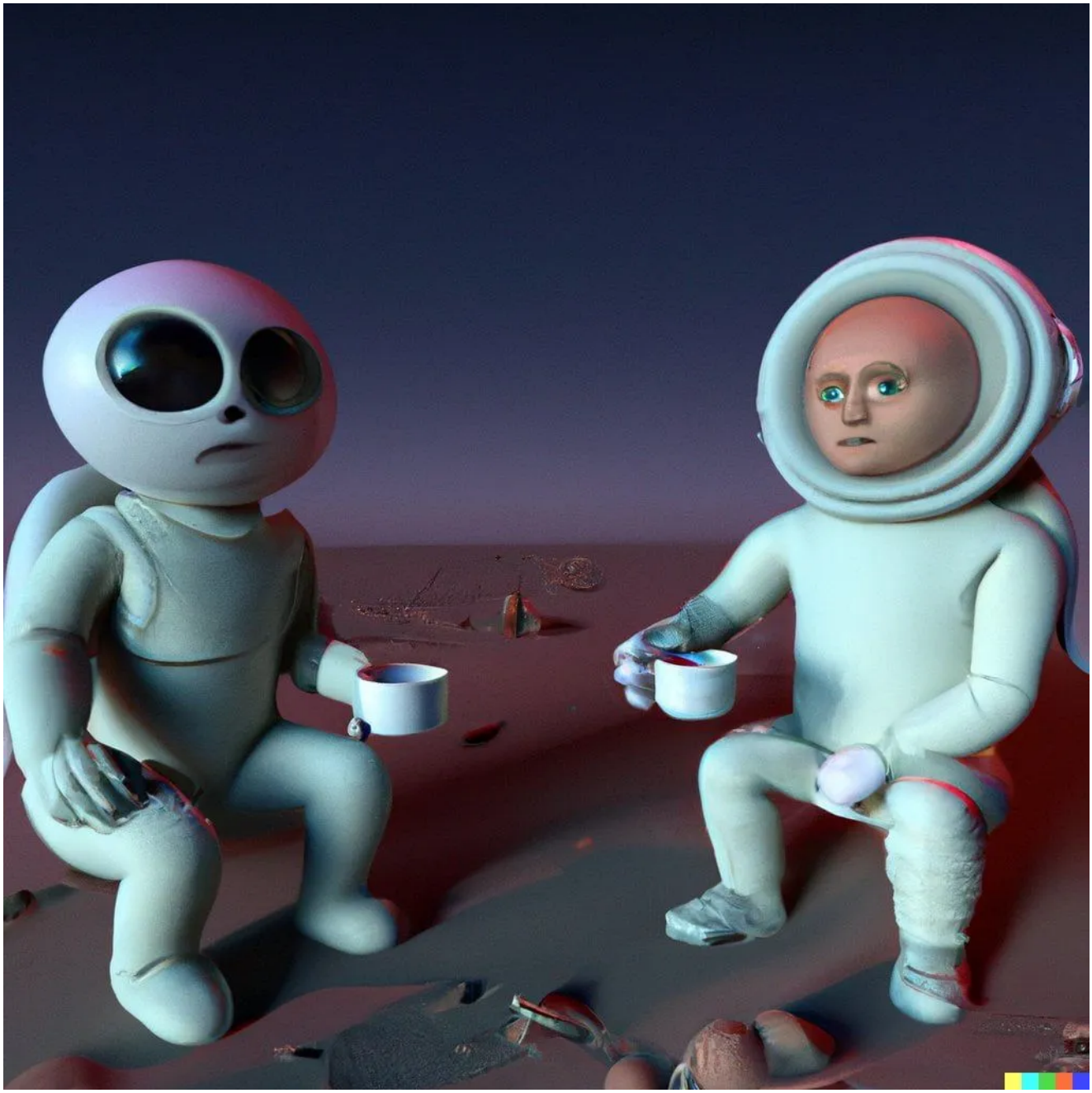
ChatGPT—What Is It? How to Use It? (Free AI Assistant)



ChatGPT is the newest Artificial Intelligence-based language model developed by OpenAI. Essentially, ChatGPT is an AI-based chatbot that can answer any question. It understands complex topics, like...

Continue Reading

13 Best AI Art Generators of December 2022 (I Tried Them All)



Choosing the right type of AI art generator is crucial to produce unique, original, and professional artwork. With the latest advancements in AI art generation, you can...

Continue Reading

Newor Media Review: Is It the Best AdSense Alternative?

Image Credit: Newor Media To turn yourself into a full-time blogger, you have to be good at monetizing your blog. One of the predominant approaches to monetizing...

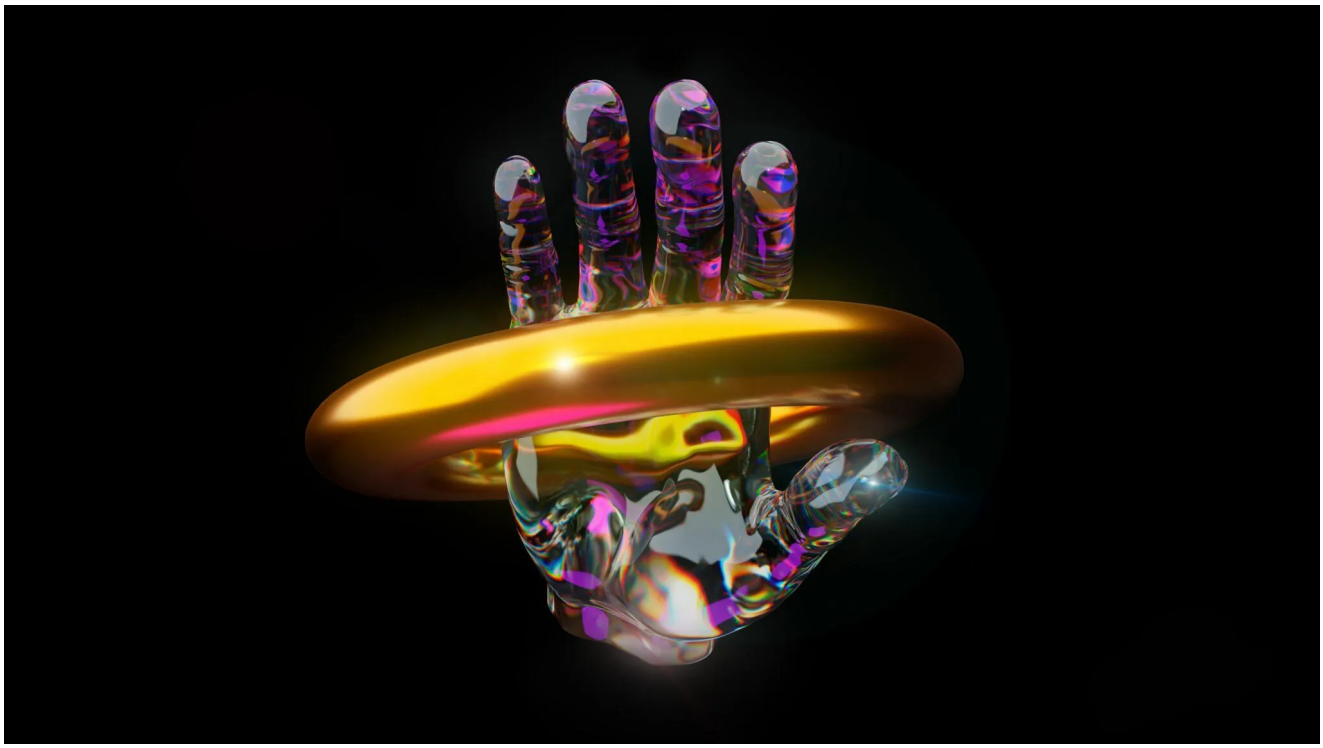
Continue Reading

How to Make an App — A Complete 10-Step Guide (in 2023)

Are you looking to create the next best-seller app? Or are you curious about how to create a successful mobile app? This is a step-by-step guide on...

Continue Reading

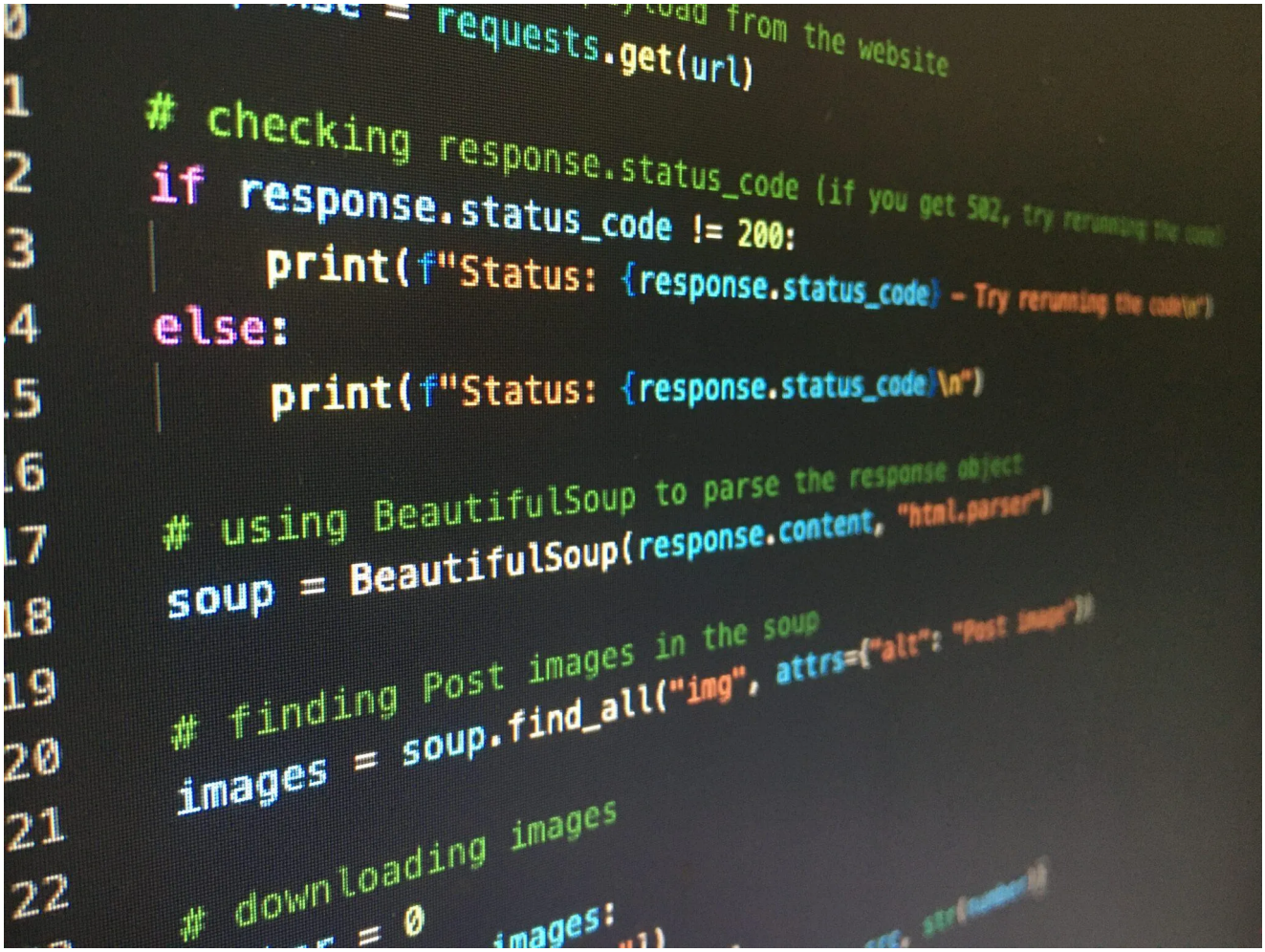
9 Best Graphic Design Courses + Certification (in 2023)



Do you want to become a versatile and skilled graphic designer? This is a comprehensive article on the best graphic design certification courses. These courses prepare you...

Continue Reading

8 Best Python Courses with Certifications (in 2023)



Are you looking to become a professional Python developer? Or are you interested in programming but don't know where to start? Python is a beginner-friendly and versatile...

Continue Reading

Recent Posts

ChatGPT—What Is It? How to Use It? (Free AI Assistant)

13 Best AI Art Generators of December 2022 (I Tried Them All)

4 Ways to Check If a String Contains a Substring in JavaScript

Python How to Check If an Object Has an Attribute (hasttr, getattr)

3 Ways to Remove a Property from a JavaScript Object

Categories

Artificial Intelligence

Crypto & NFT

Data Science

Favorites

Git

iOS Development

JavaScript

Linux

Programming

Programming Tips

Python

Python for Beginners

R

Software

Swift

Swift for Beginners

Technology

Web development